Signals_definition

Сигналы

являются простейшим, но мощным средством межпроцессного взаимодействия и передают уведомления о некоторых происходящих событиях.

Событие, порождающее сигнал, может быть вызвано

- действием пользователя,
- другим процесом,
- ядром.

Действия, вызываемые для обработки сигналов, являются принципиально **асинхронными**.

Signals_appearance

Сигнал может быть **отправлен**

- нажатием соответствующей комбинации клавиш,
- системным вызовом **kill**(PID, number).

Каждый сигнал имеет свой **номер** и соответствующее ему имя (**мнемонику**).

Linux использует, более чем **60** разновидностей сигналов.

Чтобы увидеть все виды сигналов (мнемоники и номера):

kill -l или **trap -l**

Signals_examples

Название	***Действие***	***Значение***
SIGINT	**Завершить**	Сигнал посылается ядром всем процессам текущей группы (<Ctrl> или <Ctrl>+<C>).
SIGKILL	**Завершить**	Сигнал, при получении которого выполнение процесса завершается. Нельзя ни перехватить, ни игнорировать.
SIGQUIT	**Завершить +соге**	Сигнал посылается ядром всем процессам текущей группы при нажатии <Ctrl>+<\>.
SIGSTOP	**Остановить**	Сигнал отправляется всем процессам текущей группы при нажатии <Ctrl>+<Z>. Останов выполн процесса.
SIGUSR1	**Завершить**	Сигнал предназначен для прикладных задач, как простейшее средство межпроцессного взаимодейств.
SIGUSR2	**Завершить**	Сигнал предназначен для прикладных задач, как простейшее средство межпроцессного взаимодейств.

Signals_handling

В число ситуаций при которых ядро отправляет процессу (или целой группе) определенные сигналы, входят и так называемые аппаратные особые ситуации. Например деление на ноль, обращение к недопустимой области памяти и др.

В зависимости от типа сигнала (события его породившего), реакцией системы на его получение (обработкой **on default**) может быть:

- **Exit** – выполнение действий совпадающих с семантикой системного вызова exit();
- **Core** – создание файла снимка текущего состояния ядра, затем выполнение exit(); файл core, хранящий образ памяти процесса, может быть впоследствии проанализирован программой отладчиком для определения состояния процесса непосредственно перед завершением;
- **Stop** – остановка и подвешивание процесса;
- **Ignore** – игнорировать, отбросить сигнал.

Signals_handler

Вместо того, чтобы предоставлять обрабатывать сигнал системе, процесс может обладать **собственным обработчиком**, выполняющим при получении сигнала какие-либо специфические действия.

Системный вызов **signal**() инициализирует собственный обработчик (устанавливает диспозицию) сигнала.

- **Первым** параметром вызова указан номер сигнала.
- **Второй** параметр вызова signal() указывает на собственный обработчик этого сигнала.

Программа *signal_catch* демонстрирует функционирование собственных обработчиков двух сигналов **SIGINT** (отправляем нажатием Ctrl-C) и **SIGQUIT** (отправляем нажатием Ctrl-\).

Signal_catch

```
/* Программа signal_catch.cpp */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<signal.h>
main(void)
{
int    i;
void  signal_catcher(int);
if(signal(SIGINT, signal_catcher)==SIG_ERR){
   perror("SIGINT");
   exit(1);
   }
if(signal(SIGQUIT, signal_catcher)==SIG_ERR){
   perror("SIGQUIT");
   exit(2);
   }
for(i=0; ;++i){          /* Зацикливание */
   printf("%i\n", i);    /* Индикация счетчика */
   sleep(1);
   }
}
```

Signals_handler

```
void signal_catcher(int the_sig){
    signal(the_sig, signal_catcher);  /* Сброс */
    printf("\nSignal %d received.\n", the_sig);
    if(the_sig==SIGQIUT)
    exit(3);
}
```

Системный вызов **signal**() своим вторым параметром может устанавливать диспозицию сигнала не только на функцию-обработчик, но и

- на **SIG_IGN** игнорирование сигнала ,
- на **SIG_DFL** обработку по умолчанию.

При входе в функцию-обработчик диспозиция перехваченного сигнала автоматически сбрасывается on default action. Чтобы сохранить диспозицию на функции-обработчике в ней снова настраивают перехват (см. /* Сброс */).

Сигналы **SIGKILL** и **SIGSTOP** не могут быть перехвачены или игнорированы.

Signal_alarm

```
/* Программа signal_alarm.cpp */
/* Иллюстрация использования SIGALRM, setjmp и longjmp */
/* для воплощения майм-аута */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<setjmp.h>
#include<signal.h>
main(void)
{
char    buffer[100];
int     v;
while(1){
   printf("enter a string:");
   v = t_gets(buffer, 5);
   switch(v){
         case -1:        exit(1);/* Возможно EOF */
         case -2:        printf("timed out!\n");
         break;
         default:        printf("you typed %d caracters\n", v);
         }
   }
}
```

Signal_alarm

```
jmp_buf  timeout_point;
                    /* Это обработчик сигнала SIGALRM */
void timeout_handler(int sigtype)
{
longjmp(timeout_point, 1);
}
   /* This is the important bit */
int t_gets(char *s, int t)
                    /* Буфер для значения тайм-аута в секундах */
{
char *ret;
signal (SIGALRM, timeout_handler);
if(setjmp(timeout_point) != 0)
   return -2;                       /* Тайм-аут */
alarm(t);
ret = gets(s);
alarm(0);       /* Снять звуковой сигнал */
if (ret==NULL) return -1;      /* EOF */
   else return strlen(s);
}                   /* Завершение  ^D */
```

InterProcessCommunications

В системе **IPC** Linux особое значение имеют три технологии:

- очереди сообщений (Message Queue),
- семафоры (Semaphores),
- разделяемая память (Shared Memory).

Общее:

- Объекты IPC используются совместно произвольными процессами.
- Остаются существовать в системе даже после завершения этих процессов.
- Процедура назначения имен объектам IPC нетривиальна.
- Каждый объект имеет свой уникальный идентификатор (дескриптор).
- Уникальность дескриптора обеспечивается внутри типа объектов IPC.
- Работа со всеми 3-мя видами средств в определенной степени унифицирована.

IPC_ftok() unification

Имя для объекта IPC называется ключом **key**
и генерируется функцией **ftok**():

```
#include <sys/types.h>
#include <sys/ipc.h>
...
key_t ftok (char *filename, char proj);      // filename - имя некоторого файла,
                                            // известного всем взаимодействующим
        // процессам (стабильного).            proj - идентификатор проекта.
```

Унификация работы с IPC:

- ✔ обобщенный вызов **get**() для создания объекта (получения доступа),
- ✔ флаги создания объекта **ipcflag**
- ✔ обобщенный вызов **ctl**() для управления объектами.

IPC_get() ipcflag ctl()

msget(), **msgctl**() - message queue

semget(), **semctl**() - semaphores

shmget(), **shmctl**() - shared memory

Переменная **ipcflag** определяет **права доступа** к объекту и указывает

- создается ли новый объект ***IPC_CREAT*** или
- требуется доступ к существующему ***IPC_EXCL***.

Операции над созданными объектами IPC (помещение/получение сообщения, установка семафоров, чтение/запись в разделяемую память) производятся с помощью других системных вызовов, также **унифицированных**.

IPC_manage

- Для каждого из созданных IPC объектов ядро операционной системы поддерживает внутреннюю *системную структуру* данных.
- Управляются поля структуры вызовами типа **ctl**().
- Операционная система не удаляет созданные объекты IPC даже, когда ни один процесс не пользуется ими.
- Удаление созданных объектов IPC дело самих процессов. Которые должны “договориться” об этом.

MessageQueue_description

- Процессы могут обмениваться через **mq** структурированными данными, имеющими следующие атрибуты:
  - тип сообщения (мультиплексирование разных сообщений в одной **mq**),
  - длина сообщения в байтах,
  - собственно данные (могут быть структурированы).
- Процессы могут записывать и считывать сообщения из разных **mq**.
- Процесс, отправивший сообщение в **mg**, может не дожидаться чтения его другим процесом,
  а просто завершиться, оставив сообщение в **mq**.

MessageQueue_generation

В примере создается 5 очередей сообщений, затем вызовом **popen**() выполняется shell команда **ipsc** , после этого все очереди удаляются.

```
/* Программа gener_mq.cpp */

/* Создание очереди сообщений */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX 5
```

MessageQueue_generation

```
main(void){
FILE *fin;
char  buffer[PIPE_BUF];
char  u_char = 'A';
int    i, n, mid[MAX];
key_t key;
for (i=0; i<MAX ; ++i, ++u_char){
   key = ftok(".", u_char);    /* Генерация ключа и создание ресурса */
      if ((mid[i] = msgget(key, IPC_CREAT | 0660))==-1){
      /* IPC_CREAT – создавать новую, даже если msg уже имеется */
         perror("Queue create");
         exit(1);
      }
   }
fin = popen("ipcs", "r");       /* Запуск ipcs команды */
while((n = read(fileno(fin), buffer, PIPE_BUF))>0)
   write(fileno(stdout), buffer, n);
                           /* Вывод команды ipcs */
   pclose(fin);
for (i=0; i<MAX; ++i)
   msctl(mid[i], IPC_RMID, (struct msgid_ds *)0);
                           /* Удаление */
   exit(0);
}
```

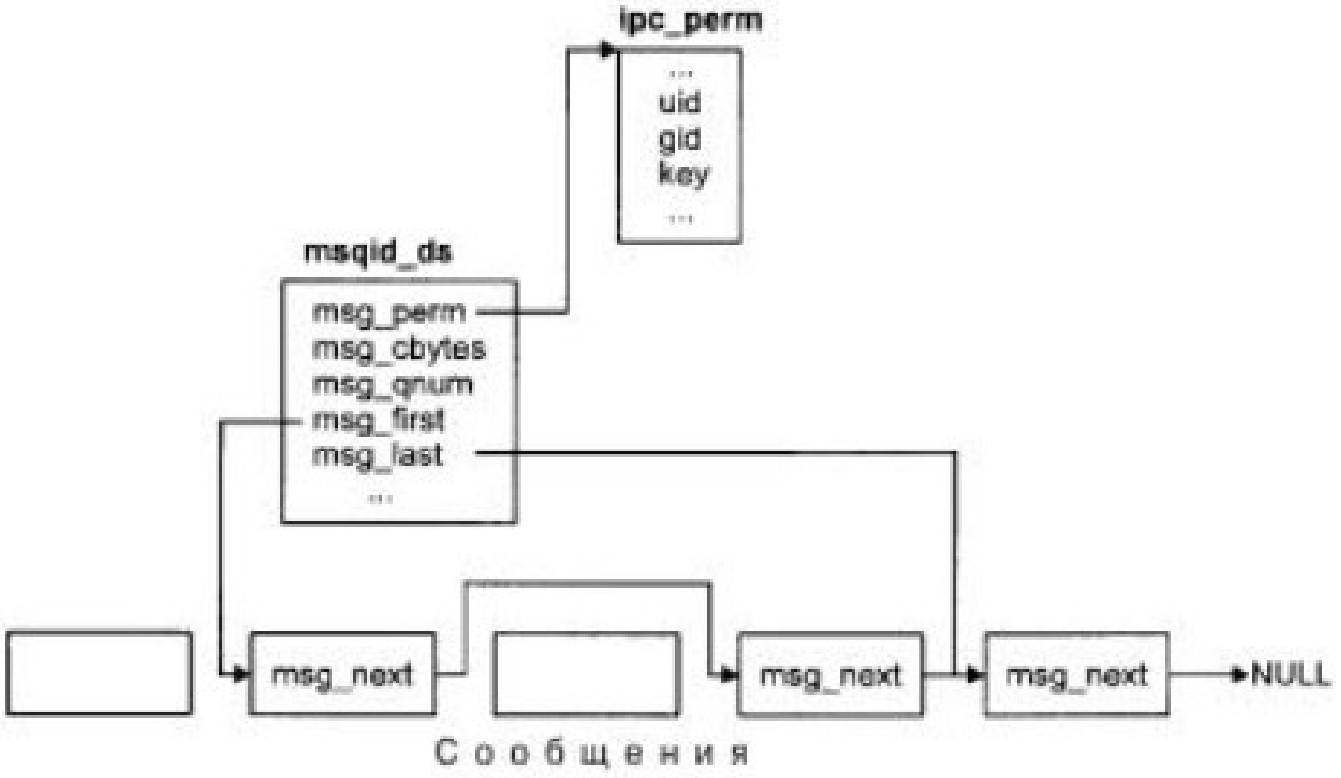
MessageQueue_structure

В адресном пространстве ядра очередь сообщений хранится в виде **однонаправленного связанного списка**.

Для каждой очереди ядро создает **msgid_ds**, заголовок с информацией:

- **msg_perm** о правах доступа к очереди,
- **msg_cbytes** число байтов и **msg_qnum** число сообщений в очереди,
- **msg_first** указатели на первое и **msg_last** на последнее сообщения.

MessageQueue_msgid_ds



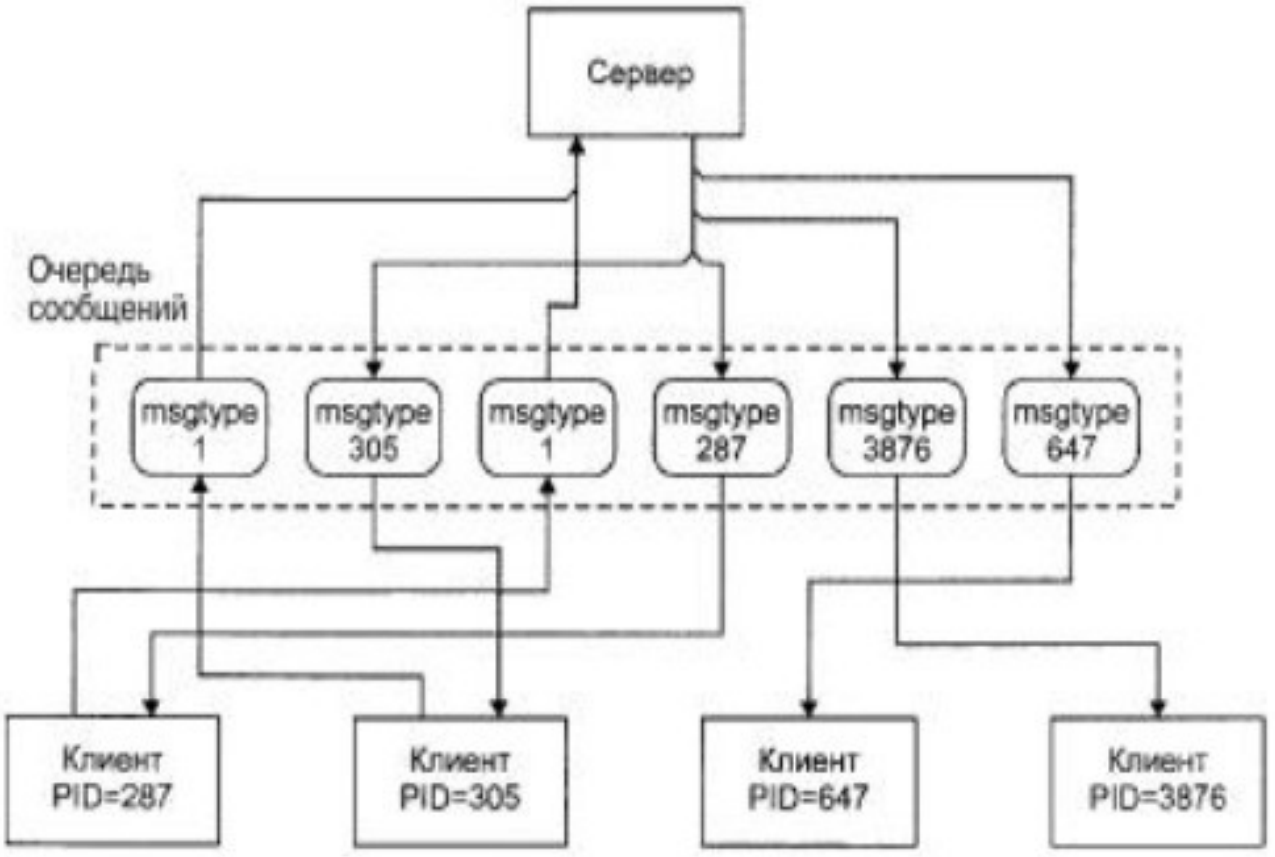
MessageQueue_commands

После создания очереди сообщений процессы получают возможность коммуникации посредством системных вызовов:

- **msgsend**() - поместить сообщение в очередь,
- **msgrcv**() - получить сообщение,
- **msgctl**() - управление сообщениями.

В очереди мультиплексируются сообщения от различных процессов, для демультиплексирования используется атрибут **msgtype**, на основании которого процесс фильтрует сообщения из очереди с помощью вызова **msgrcv**().

MessageQueue_client server communicate



MessageQueue_local include

```
/* Файл   mq_local.h */
/*
 * Общий заголовочный файл для примера программы
 *        Message Queue Client-Serve
 */
 #include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <errno.h>
#define SEED  'g'       /* Заготовка для  ftok  */
#define SERVER 1L    /* Сообщение для сервера */

typedef struct {
  long        msg_to;
  long        msg_fm;
  char        buffer[BUFSIZ];
}MESSAGE;
```

MessageQueue_client

```
#include "mq_local.h"
main(void){
   key_t       key;              /* Ключевое значение для ftok */
   pid_t       cli_pid;          /* Идентификатор процесса Process ID */
   int         mid, n;           /* Идентификатор очереди сообщений Message queue ID  */
   MESSAGE     msg;              /* Структура сообщения */
   static char   m_key[10];      /* Для символьной версии Message queue ID */
 cli_pid  =  getpid();
   if ((key = ftok(".", SEED)) == -1) {         /* Генерация ключа */
     perror("Client: key generation");
     exit(1);
   }
                /* Создание очереди сообщений и получение доступа  */
   if ((mid = msgget(key, 0)) == -1)  {
       mid = msgget(key, IPC_CREAT | 0660);
        switch   (fork()) {
        case -1:
            perror("Client: fork");
            exit(3);
        case  0:
            sprintf(m_key, "%d", mid);  /* Перевод в строку символов */
            execlp("servermq.out", "servermq.out", m_key, "&", 0);
            perror("Client: exec");
            exit(4);
        }
   }
```

MessageQueue_client

```
while (1) {
  msg.msg_to = SERVER;              /* Тип сообщения */
  msg.msg_fm = cli_pid;             /* Связывание с PID клиента */
  write (fileno(stdout), "cmd>", 6); /* Подсказка */
  memset(msg.buffer, 0x0, BUFSIZ);        /* Очистка буфера */
  n = read(fileno(stdin), msg.buffer, BUFSIZ);
  if (n == 0)                       /*  EOF ? */
    break;
  if (msgsnd(mid, &msg, sizeof(msg), 0) == -1)  {
    perror("Client: msgsend");
    exit(5);
      }
  if ((n = msgrcv(mid, &msg, sizeof(msg), cli_pid, 0)) != -1)
    write(fileno(stdout), msg.buffer, strlen(msg.buffer));
      }
  msgsnd(mid, &msg, 0, 0);
  exit(0);
}
```

MQ_server

```
#include "mq_local.h"
main(int argc, char *argv[]) {
  int       mid, n;
  MESSAGE    msg;
void       process_msg(char *, int);
    if (argc != 3) {
    fprintf(stderr, "Usage: %s msq_id &\n", argv[0]);
    exit(1);
  }
  mid = atoi(argv[1]);          /* Идентификатор очереди сообщений */
                                /*       как параметр командной строки */
  while (1) {
    if ((n = msgrcv(mid, &msg, sizeof(msg), SERVER, 0)) == -1) {
      perror("Server: msgrcv");
      exit(2);
    } else if (n == 0)                  /* Клиент отработал */
      break;
     else {                              /* Обработка сообщений */
       process_msg(msg.buffer, strlen(msg.buffer));
       msg.msg_to = msg.msg_fm;/* Свопинг сообщений:  to <-> from */
       msg.msg_fm = SERVER;
       if (msgsnd(mid, &msg, sizeof(msg), 0) == -1) {
         perror("Server: msgsnd");
         exit(3);
           }
        }
  }                                     /* Удаление очереди сообщений */
   msgctl(mid, IPC_RMID, (struct msqid_ds *) 0);
   exit(0);
}
```

MessageQueue_server

```
        /* Перевод строчных символов сообщения в прописные */
void  process_msg(char *b, int len) {
   int     i;
   for (i = 0; i <len; ++i)
     if (isalpha(*(b + i)))
       *(b + i) = toupper(*(b + i));
}
```

Thanks for your attention

Спасибо за внимание !

vladimir.shmakov.2012@gmail.com